An Approach to Manage Reconfigurations and Reduce Area Cost in Hard Real-Time Reconfigurable Systems

This article presents a methodology to build real-time reconfigurable systems that ensure that all the temporal constraints of a set of applications are met, while optimizing the utilization of the available reconfigurable resources. Starting from a static platform that meets all the real-time deadlines, our approach takes advantage of run-time reconfiguration in order to reduce the area needed while guaranteeing that all the deadlines are still met. This goal is achieved by identifying which tasks must be always ready for execution in order to meet the deadlines, and by means of a methodology that also allows reducing the area requirements.

Categories and Subject Descriptors: C.0 [**Computer Systems Organization**]: General – *Systems specification methodology*; C.1.3 [**Processor Architectures**]: Other Architectures Styles – *Adaptable Architectures*; C.3 [**Special-Purpose and Application-Based Systems**] – *Real-time and Embedded Systems*
General Terms: Algorithms, Design, Performance
Additional Key Words and Phrases: Reconfigurable architectures, FPGAs, real-time task scheduling, embedded systems design

1. INTRODUCTION

New-generation embedded systems demand features such as high performance, reduced area and energy/power efficiency. Hardware-based solutions are normally used as accelerators that provide better performance and less energy consumption than purely software ones, since they eliminate the overhead due to instruction decoding and they include optimized hardware for the requested operations. However, due to the exigent area restrictions and the complex and expensive design process, it is not always feasible to implement all the needed functionalities of an embedded system using only a hardware solution based on Application Specific Integrated Circuits (ASICs).

Reconfigurable hardware is a very interesting technology to build these accelerators (So, H. K.-H. et al. [2008a], [2008b], Chang, C. et al. [2005]). It features important properties such as high performance, since optimized circuits can be designed for each task; flexibility, since the functionality of the hardware can be modified at run time to match the platform requirements; area savings, since the same reconfigurable area can be used to execute different tasks that do not demand concurrent execution; and faster time-to-market compared to ASIC solutions. For these reasons, this technology is becoming more and more important for industry, especially Field Programmable Gate Arrays (FPGAs), where nowadays Xilinx™ and Altera® dominate the market (OPENCORES [2011]).

However, one of the main drawbacks of reconfigurable hardware is the large reconfiguration latency, which can be of the order of hundreds of milliseconds (XILINX, [2012b]). Applying partial reconfiguration can largely reduce this latency; i.e., reconfiguring only a region of the FPGA, while the remaining area remains unaltered. However, many systems with real-time constraints still consider this latency unacceptable even if it is reduced to just a few milliseconds.

In this paper we propose a methodology to develop reconfigurable systems that can execute a given set of applications guaranteeing that all their real-time constrains are met. In this research, we have worked with applications represented as *Directed Acyclic Graphs* (DAGs, Kavi, K. et al. [1986]), which nodes represent computational tasks, and which edges represent data dependencies between two tasks. This is a very common representation for applications in embedded systems. In addition, the set of task graphs that the system must execute is known at design time.

The proposed methodology analyzes the task graphs to be targeted onto the reconfigurable device and computes the size of the reconfigurable architecture needed, in order to execute all of these applications altogether while meeting all their deadlines. This methodology has been designed for hard real-time systems. Hard real-time scheduling involves guaranteeing that the system will *always* meet all the deadlines, whereas soft real-time scheduling allows for a certain task rejection rate. Both kinds of real-time systems are everywhere around us. They are used for any system that deals with critical issues, such as security. We can illustrate the difference between them with an MPEG player application: if this player is used on a computer to watch a film, it will not be a big deal if at the beginning the screen does not display the image properly for a few seconds. This is an example of a soft real-time system. However, if the player is used as by an artificial vision system that recognizes patterns in order to drive a remote device, that delay is unacceptable and may involve a security issue (in this case, images must be always processed at the proper time or the device may crash). This is an example of a hard real-time system.

The rest of the paper is organized as follows: The remainder of Section 1 describes our target architecture (Subsection 1.1), introduces our methodology by means of an example (Subsection 1.2), states the problem at hand and the assumptions upon which we have worked (Subsection 1.3) and enumerates the benefits of the proposed approach (Subsection 1.4). Next, Section 2 overviews other relevant works on reconfiguration overhead elimination and Section 3 describes the proposed methodology. Section 4 presents the experimental results and finally, Section 5 summarizes this article with final conclusions and possible lines of future work.

1.1 Target Architecture

The approach proposed in this paper is applicable to any hardware device that supports dynamic reconfiguration, as long as the size of the reconfigurable area can be customized in order to fit the requirements of the system. In our work, we have adopted the reference hardware architectural model shown in Figure 1.

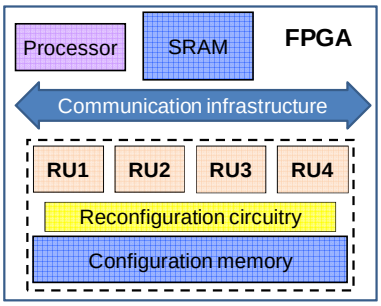

Fig. 1. Target architecture.

This architecture comprises a static region (which functionality does not change at run time) and a dynamically reconfigurable region (highlighted in Figure 1 by means of the dashed line).

This platform is a microprocessor-based system that contains at least one *Processor*, which manages the general operation of the system; a *SRAM* memory, which stores the instructions and the data; and a set of elementary reconfigurable regions named Reconfigurable Units (RUs in the figure), which represent the smallest amount of area that can be reconfigured at a time. All these cores are connected among them by means of

the *Communication infrastructure* (it could be implemented as a system bus or as a Network-on-a-Chip).
All the RUs are wrapped with a fixed interface that provides the basic operating system (OS) and communication functionality. This does not appear in the figure for simplicity. With this support, each RU can independently execute a task, and communicate with the other processing elements with similar communication latencies (since they are connected through the same communication infrastructure). Although the system includes several RUs, we assume that only one of them can be reconfigured at a time, since the *Reconfiguration Circuitry* is shared for all the reconfiguration resources, as it happens in current commercial platforms. Finally, and as in all the FPGA-based systems, the configurations are fetched from the *Configuration memory*, which can be both on-chip and off-chip.
Such a system can be implemented on last-generation FPGAs, such as Xilinx™ Virtex-5 and Virtex-6 devices (XILINX [2012b], [2012c]), or Altera™ ones (ALTERA [2011a], [2011b]). To this end, vendors provide specific design tools to develop custom SoCs. On the one hand, the Xilinx™ EDK development tool (EDK [2010]) can be used to develop a processor-based system; for instance, using the MicroBlaze soft-processor (XILINX [2012a]). In the latest versions of this software, several options for the communication infrastructure are possible: For instance an IBM® Processor Local Bus (PLB, IBM® Microelectronics [1999]), is frequently used to attach the computational cores; and the LMB bus (XILINX [2005]), for the memories. The RUs can be implemented as peripherals, and their dynamically partial reconfiguration can be easily managed by using the Plan Ahead tool (XILINX [2009]). On the other hand, the Altera™ Quartus-II software (ALTERA [2011b]) allows the development of systems based on the Nios® II soft processor core (ALTERA [2011c]).
In any case, it is very important to underline that the methodology proposed in this article does not rely on the specific FPGA that is finally used or on the final implementation of the system, as long as it follows the architectural model depicted in Figure 1. Thus, we assume that the designer team has already tested the architecture, in order to be sure that it fulfills the system requirements (this will be explained in Subsection 1.3).

1.2. Motivational Example

The example of Figure 2 shows the execution of two task graphs in a system with certain number of RUs. Our objective is to hide 100% of the delays due to the dynamic reconfigurations of the tasks while using the minimum number of RUs. In this example the average execution time of a task is 5.3 milliseconds, and we assume that the reconfiguration latency is 4 milliseconds. This is a very demanding scenario since there is a very small margin to hide these latencies.
The ideal execution times of these task graphs are 15 milliseconds (Figure 2.d). However, if no active policy to reduce the reconfiguration overhead is included in the system, the actual execution time taking into account the reconfigurations is 27 and 23 milliseconds respectively, as it is depicted in Figure 2.a.
These execution times can be greatly reduced by applying a prefetch technique, such as the one that was proposed by Li, Z. et al. [2002], and Hauck, S., [1998], which carries out some reconfigurations in advance, as depicted in Figure 2.b. However, a prefetch technique alone cannot achieve the objective of hiding 100% of the reconfiguration latency, since in this example there is not any margin to hide the latency of the reconfigurations of Tasks *a* and *d*.
The optimizations that we propose applying to these schedules consist on guaranteeing that these two tasks will be always reused, while minimizing the number of RUs needed. In other words, when Task Graphs 1 or 2 are executed, Tasks *a* and *d* will be always present in one of the RUs. In addition, since the reconfiguration latency of the remaining

tasks can be hidden with the prefetch technique, the result is that the reconfigurations of these two task graphs will not generate any additional delays.

In order to apply these optimizations, we need to identify those tasks which reconfigurations cannot be hidden even when a prefetch technique is applied. We name these tasks *Compulsory-Reuse (CR)* tasks. As this name indicates, these tasks must always be reused in order to meet the given deadlines. A possible way to achieve this objective is by statically assigning them to a set of RUs, and never loading any other task on those RUs, as it is depicted in Figure 2.c. In this case we need four RUs, two for Tasks *a* and *d*, and two additional ones that are shared among the other tasks. With this solution we already succeed at hiding all the reconfiguration latencies.

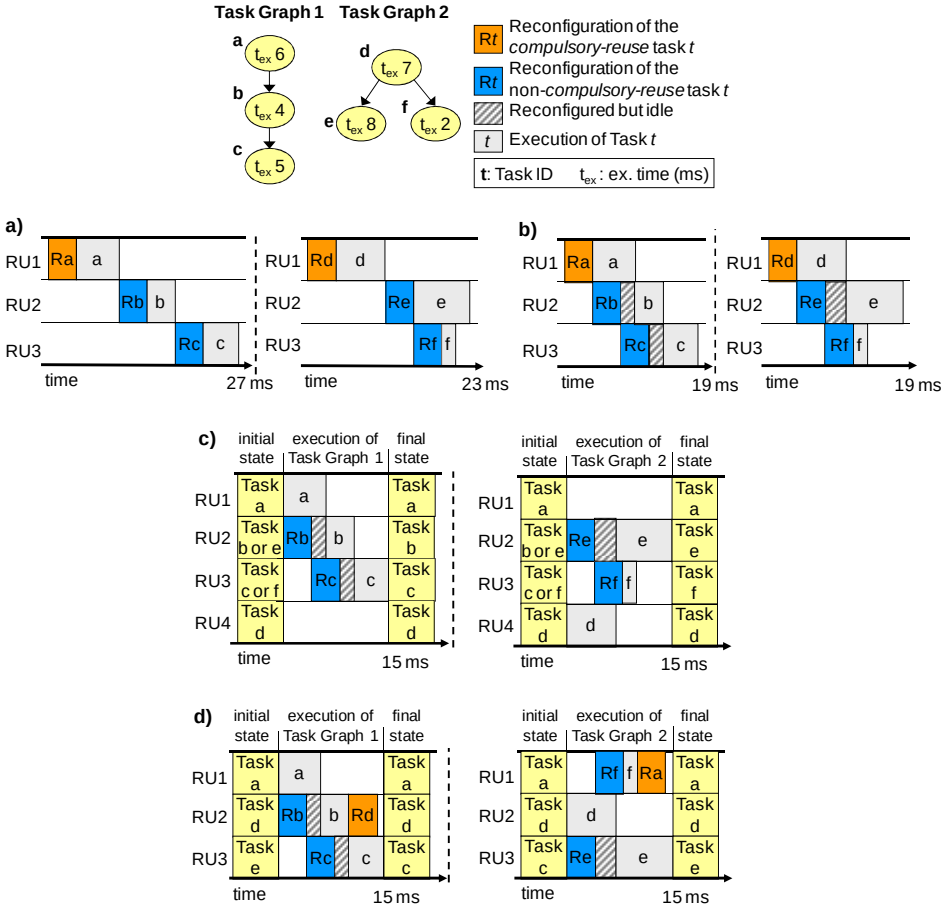

Fig. 2. (a) Sub-optimal schedules for both task graphs assuming that the reconfiguration latency is always 4 milliseconds. RU n: reconfigurable unit n. (b) Schedules applying an optimal prefetch approach. (c) Schedules that guarantee no reconfiguration delays for 4 RUs. (d) Schedules that guarantee no reconfiguration delays for the minimum possible number of RUs.

However, we also want to minimize the number of needed RUs. To this end, we propose to *modify the original schedules* in such a way that the *CR* tasks already loaded in the system can be replaced by other tasks as long *as they are loaded back again before the end of the execution of the current task graph*.

This idea is illustrated in Figure 2.d. In this case, RUs 1 and 2 are assigned for the execution of the *CR* tasks *a* and *d*, respectively; whereas RU3 is assigned for the execution of Tasks *c* and *e*. Finally, Tasks *b* and *f* are again scheduled in RUs 2 and 1, hence they replace Tasks *d* and *a* respectively. However, these *CR* tasks are loaded back before the end of the execution of the involved task graphs in order to guarantee that they are always loaded in the system when the following task graph starts its execution. The figure presents the two possible worst-case scenarios: on the left side, it depicts the execution of Task 1 when the previous executed graph was Task Graph 2; whereas on the right side it depicts the execution of Task Graph 2 when Task Graph 1 was previously executed. As the figure shows, even in these worst-case scenarios, this approach hides 100% of the reconfiguration latency while using only three RUs.

1.3. Assumptions and Problem Statement

The objective of this section is to discuss the assumptions upon which we have based this research work, as well as to state the problem addressed in this paper.
Figure 3 presents an overview of the different steps needed to design a reconfigurable real-time system. The work presented in this paper only focuses on the last step, the *Run-Time Reconfigurations Management*, and assumes that the previous steps have already been carried out.

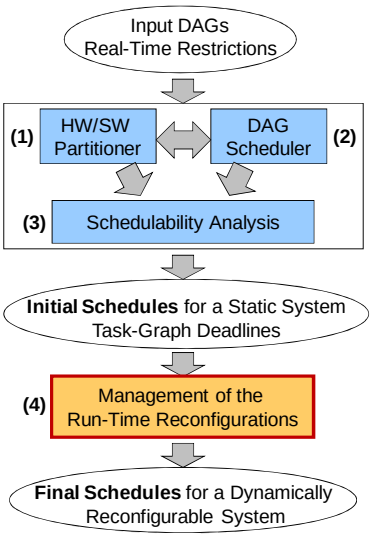

Fig. 3. Flowchart of the system design methodology.

The input of this flow is a set of applications, which are represented as Directed Acyclic Graphs (*Input DAGs* in the figure) and a set of *Real-Time Constraints* associated to them. Since we are targeting real-time sytems, we assume that each task in these DAGs has an accurate worst-case estimation of their execution time, since otherwise it would not be possible to guarantee that the deadlines are met. We are also assuming unidirectional point-to-point communications.
With this information, the *HW/SW Partitioner* (Step 1) and the *DAG Scheduler* (Step 2) firstly interact between them to decide which task graphs are executed in reconfigurable hardware, and to assign an *Initial Schedule* to each one of them. Then, the *Schedulability Analysis* (Step 3) assigns a *Deadline* to each task graph in such a way that if all the deadlines are met, the system will met the real-time restrictions. These three steps

schedule the task graphs assuming that only one task graph is executed at a time. In other words, the task graphs are executed sequentially, not concurrently. Executing several task graphs in parallel is only possible if the opportunity of executing these tasks in parallel is known at design-time (in this case it is enough to merge all these task graphs in order to generate a new one including all of them). This is a common assumption for real-time systems, since if we allow that several graphs can be concurrently executed, they may incur into conflicts for the use of the shared reconfigurable resources, as well as the system communication infrastructure or the shared memory. Hence, their execution times will be unpredictable at design time and it will not be possible to guarantee that the run-time constraints are met. If more than one task graph is waiting for execution, a task-graph scheduler will select at run-time the execution order based on some criteria, such as their deadlines or their priorities. This criteria is known at design-time, and it has been taken into account during the schedulability analysis in order to guarantee that the deadlines are always met. A relevant reference about this process has been presented by Sprunt, B. et al. [1989].

The additional step that we propose (Step 4 in Figure 3) uses run-time reconfiguration in order to reduce the hardware requirements while guaranteeing that all the tasks meet the deadlines imposed by the schedulability analysis.

Since initially the number of RUs that the system will need is unknown, for each task graph, the *Initial Schedules* assign each hardware task to a different RU. These schedules assume that the hardware tasks are always loaded in the system; i.e., they assume that they run in a static system with no run-time reconfiguration. Then, the proposed *Management of the Run-Time Reconfigurations step processes the input task graphs, their deadlines and their initial schedules, and allows for the run-time reconfiguration in order to determine the number of RUs that the system needs, as well as their size in such a way that the resources consumption is optimized, while ensuring that all the task-graph deadlines are met.* For this purpose, it updates the *Initial Schedules* and returns a set of *Final Schedules* that are suitable for dynamically reconfigurable systems.

All these steps are carried out at design-time. The reason is that hard real-time systems must guarantee that all the real-time constraints of a given set of tasks are always met, and the only way to do that is to have all the information of the tasks at design-time when the schedulability analysis is carried out. Including new tasks at run-time is only possible for systems that follow a best-effort approach; i.e., systems that attempt to execute the incoming task set as fast as possible, but they cannot guarantee that all the task deadlines are met. Hence the flow depicted in Figure 3 (including our optimization techniques) is entirely carried out off-line. However, what is unknown at design time is the order of execution of these task graphs, as well as how many times each one of them will be executed in the system. The system must be ready to meet the deadlines for all the possible sequences of execution. This is typically done by guaranteeing that the system meets the deadlines even in the worst-case scenario.

Finally, the DAGs are executed in the hardware multi-tasking system described in Subsection 1.1, according to their final schedules. Each RU has been designed to run only one task at a time and each task can be loaded to any RU. Data among tasks are exchanged by using a shared memory approach; for instance, by using a SRAM memory module connected to the system bus. This allows carrying out the communications among tasks efficiently, since the accesses to these memories are carried out in just a few clock cycles.

The techniques proposed in this paper have been designed assuming that there is no run-time preemption in the execution of the tasks. Hence we assume that once a task graph starts its execution, it always finishes without any interruption. However, if the system supported task preemption, most of the techniques proposed in this article could still be applied. In fact, in this case the only one that must not be enabled in order to guarantee

the correct operation of the system is the optimization regarding the *Compulsory-Reuse* (*CR*) tasks that is described in Figure 2.d. The reason is that in the final schedules, the *CR* tasks replaced are loaded back at the end of the execution of the task graph. Hence if a task graph is preempted before the replaced *CR* tasks are loaded back, these *CR* tasks will not be reused in the next execution, and some deadlines may not be met.
Finally, we would like to point out that the techniques proposed in this article are heuristic because we do not want to overload the system with the computational complexity of an equivalent numeric formulation. Moreover, it is likely that the hardware-software partitioner will need to evaluate the hardware cost of many different partitions, by executing our techniques several times. Hence, in this scenario it is very important to develop techniques that provide good results as fast as possible.
In fact, in order to check the quality of our results, we have developed an equivalent version for the first step of our methodology (Section 3.1), which is the most critical step. This version applies a *branch&bound* approach in order to find the optimal best-effort solution (*branch&bound* is one of the techniques most frequently used by ILP solvers). This comparison demonstrates that our heuristic finds optimum solutions for all the task-graphs used in this article (see Section 4). This is a remarkable result, taking into account that its complexity is linear with the number of tasks. Of course, it should be possible to artificially create complex graphs where our heuristic is suboptimal, but we believe that our heuristics achieve their objective: finding good solutions fast. In any case, *the main contributions of this article are the ideas presented, and the methods that implement them are just an example that demonstrates the applicability of these ideas.*

1.4. Benefits of the Proposed Approach

The approach proposed in this paper is interesting for the following reasons:

1. This approach allows to use run time reconfiguration for real-time tasks with hard deadlines, since it can be used to guarantee that the reconfigurations will not introduce any delay.
2. It can be used to select the size of the FPGA needed to design a real-time system. This can be useful since vendors normally offer FPGAs with similar features but different reconfigurable area.
3. Since the problems addressed in this approach are orthogonal to the HW/SW partitioning of the application, any partitioner can be used in combination with the techniques proposed in this paper.
4. This approach transparently manages the run-time reconfigurations. Hence, on top of it any scheduler developed for real-time heterogeneous multiprocessor systems can also manage the execution of the RUs.

In addition, we would like to point out that although our approach relies on design-time information, it is still suitable for dynamic applications, since it only needs to know which tasks are going to be executed in the system, but not when they will be executed.

2. RELATED WORK

The task scheduling problem for reconfigurable hardware has been addressed by a number of publications reporting contributions in the academic and industrial world. However, the papers selected in this section reflect the significant aspects with respect to the proposed approach and allow an objective comparison of the benefits that can be achieved with our techniques. The articles discussed in this section present task

scheduling techniques for dynamically reconfigurable hardware, designed either for non-real-time or for real-time environments. The difference between these two environments is that in real-time systems the scheduler must guarantee that all the deadlines are always met. This involves having detailed information about the worst-case execution of each task, and not allowing any run-time interference in the task execution that may lead to a deadline violation. A non-real-time system typically follows a best-effort approach that attempts to achieve different objectives, such as optimizing the performance of a task, maximizing the throughput, or minimizing the number of deadlines missed. At the end some deadlines may be missed, but the average results will be very good. They are different worlds, with completely different scheduling solutions.

2.1. Non-Real-Time Task Scheduling

Non-real-time task scheduling for reconfigurable hardware has been massively studied in the literature. Most of these techniques, such as classical list-based scheduling policies (Noguera, J. et al. [2002], [2004]) are based on a prefetch approach (Hauck, S. [1998], Li, Z., et al. [2002]), which attempts to load the reconfigurations in advance in order to hide their latency.

All the most interesting related approaches can be classified into exact *ILP formulations* and *heuristic methods*. The former are useful if the problem to be solved is not of a great computational complexity and the behavior of the system is well known at design time. Otherwise it can be practically unsolvable even at design time. Two good examples were presented by Ghiasi, S. et al. [2004] and Sim, J.E. et al. [2009]. On the one hand, Ghiasi et al. [2004] propose an optimal scheduling algorithm for DAGs on a reconfigurable system that comprises a set of equal-sized reconfigurable tiles. However, the proposed approach is only proved to be optimal as long as the reconfiguration time of the tasks is zero. On the other hand, Sim, J.E. et al. [2009] present an algorithm that minimizes the impact of the reconfigurations of task graphs in an FPGA. This work assumes that the mapping of the tasks on the target device is already decided prior to their temporal scheduling, and that the applications targeted are just sequences of tasks (hence no parallelism is allowed).

However, ILP techniques are impractical for large instances of the scheduling problem due to its computational complexity, or if the system features certain degree of dynamism. In these cases, heuristic methods, such as the ones presented by Banerjee, S. et al. [2009], Cordone, R. et al. [2009], Pan, Z. et al [2008], Nahapetian, A. et al. [2009], Noguera, J. et al. [2002], [2004] and Clemente, J.A. et al. [2010] are preferred. The solution proposed by S. Banerjee et al [2009] aims at taking full advantage of the data parallelism for a given dynamic application by replicating the same task several times. Cordone, R. et al. [2009] present a partitioning and scheduling approach for reconfigurable hardware that attempts to minimize the overall latency of a given application for a system with a conventional processor and a given set of RUs. The technique proposed by Pan, Z. et al. [2008] proved to obtain very good results when targeting applications with dynamic behavior. Nahapetian, A. et al. [2009] present an heuristic algorithm to schedule independent tasks in heterogeneous reconfigurable resources. It is a good example of how an heuristic solution can be very close to the optimal one, and at the same time it greatly reduces its computational complexity. Noguera, J. et al. [2002], [2004] propose a scheduling flow and combine it with a replacement policy specifically designed to maximize task reuse in order to reduce the impact of the dynamic reconfigurations. And finally, in our previous work (Clemente, J.A. et al. [2010]) we developed a run-time reconfiguration manager. As the previous references, it is a best-effort approach that attempts to execute a task graph as fast as possible by applying a novel replacement policy that reduces the reconfiguration

overheads. However, since it is a best-effort approach it cannot guarantee that the deadlines are met. Hence it cannot be used for real-time systems. Moreover, it assumes that the platform has already been designed, and the number of RUs is known in advance. Therefore it cannot be used to reduce the area requirements of a system.

2.2. Real-Time Task Scheduling

Task scheduling on dynamically reconfigurable hardware under real-time constraints has also been studied in the literature. These techniques aim at minimizing the task reject rate under certain real-time constraints.

Many of these techniques target applications which execution is periodic. For instance, Danne, K.D et al. [2005], [2006], propose techniques based on the well-known Earliest Deadline First (EDF) policy to find a feasible schedule in which a set of periodic tasks meet their deadlines, but also minimizing the hardware resources consumption. Another interesting and more recent approach presented by Kooti, H. et al. [2010] proposes a mixed integer linear programming solution for periodic tasks that further improves the task schedulability of the EDF policy. These three approaches have been designed for soft real-time systems; i.e., they aim at minimizing the task rejection rate, but not at guaranteeing that 100% of the task deadlines are met.

In some cases the execution period of the tasks is unavailable. This makes necessary to use a different approach, such as the one proposed by Dittman, F. et al. [2007], which proposes a technique to deal with aperiodic tasks that run under hard real-time conditions. Another interesting technique, this time targeting soft real-time systems, is presented by Fazlali, M. et al. [2010], which has been successfully tested on real-world application workloads. These two approaches deal with independent tasks, i.e. they do not support the execution of DAGs with data dependencies among tasks, hence they target different objectives from the ones addressed in this article. Finally, real-time scheduling of aperiodic task graphs has also been succesfully applied on commercial platforms. Good examples are ReconOS, developed by Lübbers, E. et al. [2007] and CAP-OS, developed by Göhringer, D. et al. [2011]. These operating systems provide important services such as: soft real-time scheduling based on priority-based policies, hardware task mapping and efficient management of the available reconfigurable resources. However, these systems lack the ability to adapt the reconfigurable platform to the system needs in order to guarantee the 100% of the task schedulability while optimizing the resources consumption, contrarily to the approach presented in this article.

We believe that our work is compatible with these systems, since we are not developing a new OS for reconfigurable systems, but *a set of techniques that minimize the number of RUs needed while guaranteeing the schedulability of the input applications in hard real-time reconfigurable systems*. Hence our modules are, on the one hand, used by the system designers to identify the amount of reconfigurable resources needed and, on the other hand, used by the OS or the middleware to obtain a proper task schedule that meets all the deadlines. Our work can be used both for periodic or aperiodic tasks, as long as a schedulability analysis (Figure 3, Step 3) has been previously carried out.

To the best of our knowledge, the problem addressed in this article has not been previously addressed elsewhere. On the one hand, the works discussed in Subsection 2.1 are *basically best-effort approaches that attempt to reduce the execution time of a given application.* They can provide very good average performance, *but they cannot guarantee that a set of given deadlines are met*. On the other hand, *run-time reconfiguration has not been used for hard real-time systems* previously. Indeed, the systems described in this subsection either do not allow for run-time reconfiguration; they may generate some task reject rate, which is unacceptable in a hard real-time system; or they do not allow for

adapting the target reconfigurable platform depending on the system needs. The reason may be that dealing with the reconfigurations increases the complexity of the schedulability analysis, which is already a very complex problem. Our approach solves this issue by proposing orthogonal techniques that can be applied after carrying out this analysis, and that guarantee that none of the reconfigurations will introduce any additional delay.

3. THE PROPOSED METHODOLOGY

Figure 4 depicts the steps of the proposed methodology. It receives as input the set of task graphs to be executed, as well as their deadlines and their initial schedules. Firstly, it identifies which tasks cannot be loaded at run time without meeting the deadlines. We name those tasks *Compulsory-Reuse (CR) tasks* (Step 1). Then, for each task graph, it identifies which non-*CR* tasks can replace *CR* tasks from other task graphs and load them back on time in such a way that the task-graph deadline is still met. The number of non-*CR* tasks that fulfill this condition is named in the article as *loading-back factor* (Step 2). Then, it identifies the minimum number of RUs needed to guarantee that all deadlines are met (Step 3). After that, it assigns the tasks to the RUs and determines the minimum size required for each RU (Step 4). Finally it generates the final schedules for each task graph. The following subsections describe these steps in greater detail.

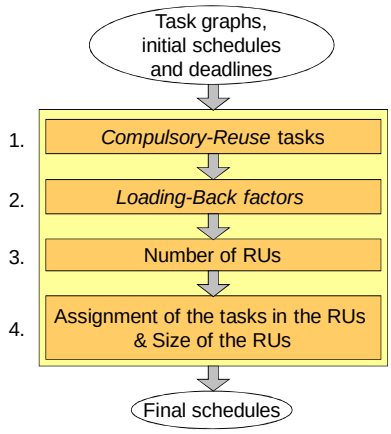

Fig. 4. Flowchart of the proposed methodology.

3.1. Identification of the "*Compulsory-Reuse*" tasks

As previously hinted, the *Compulsory-Reuse (CR)* tasks of a task graph are those tasks that introduce a delay in the execution due to their reconfigurations *unless they are already loaded* when the execution of the graph starts, and due to that delay the graph will not meet its deadline. The objective of this phase is to identify these tasks. More specifically, for each task graph, this step identifies the *minimum* set of tasks that fulfill the following condition: *If all the CR tasks are loaded at the beginning of the task-graph execution, the task graph will meet all its deadlines, in spite of the overhead due to the reconfiguration of the remaining tasks*. For simplicity, in our experiments we only include one deadline for each task graph. But they could include any additional deadline, for instance in order to guarantee that some data is sent at a given point of time, and our

techniques will still be valid. It is even possible to mark all the events in the initial schedule as time restrictions to meet, in such a way that the final schedule will be exactly the same than the initial one, but including, if possible, the reconfigurations.
Figure 5 depicts the pseudo-code of this process. For each incoming $task\_graph_i$, the algorithm firstly initializes its *CR* tasks set (*CR_set*) to the empty set (Line 2). Then, the *while* loop (Lines 3-6) iteratively invokes the *execution_time* function, in order to check if $schedule_i$ meets its $deadline_i$ (Line 3). In that case, the algorithm finishes. Otherwise the function *add_task* (Line 4) identifies which reconfiguration has generated the greatest delay and adds it to the *CR_set*. Then, it updates $schedule_i$ (Line 5) assuming that the tasks that have been added to *CR_set* are reused (hence, they do not generate any reconfiguration overhead).

```
1. FOR each task_graph_i, schedule_i, deadline_i {
2.   CR_set := Ø;
3.   WHILE (execution_time (schedule_i) > deadline_i){
4.     add_task (task_graph_i, schedule_i, &CR_set);
5.     update (schedule_i);
6.   }
7. }
```

Fig. 5. Pseudo-code of the algorithm that identifies the *Compulsory-Reuse* tasks.

The exe*cution_time* function attempts to optimize the execution time of the graph. Scheduling a task graph taking into account the reconfigurations is a complex issue. Fortunately, as explained in the *Related Work* section, several research groups have already developed good algorithms for this problem, and any of these schedulers for reconfigurable systems can be used in this step. We have selected the one presented by Clemente, J.A. et al. [2010], since it is fast and provides good results hiding most of the reconfiguration overheads with a prefetch approach. This scheduler basically assigns a weight to each task taking into account how critical that task is for the graph execution and uses these weights to decide the reconfiguration sequence. The idea is to assign greater weights to the tasks that belong to the critical path of the graph and to attempt to prefetch those tasks as soon as possible in order to prevent delays in the critical path.

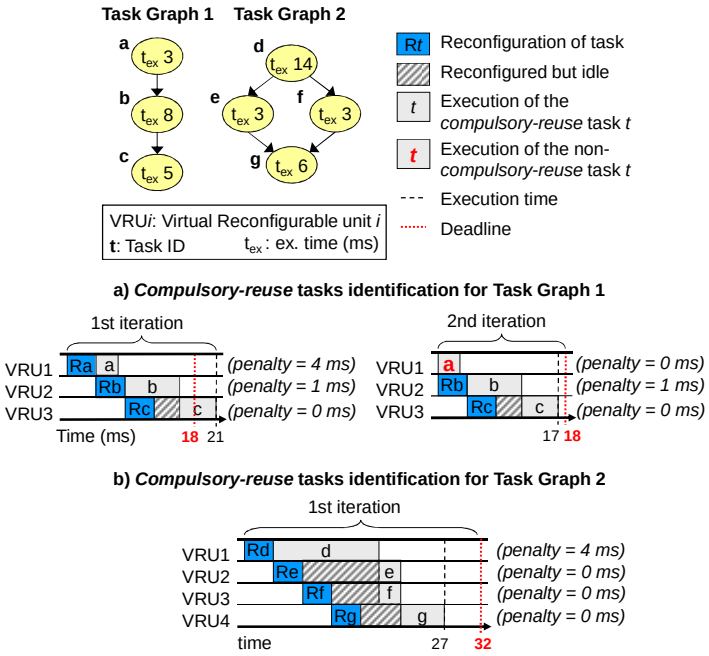

Fig. 6. Example for the *Compulsory-Reuse* tasks identification process. The reconfiguration latency is always 4 milliseconds.

Figure 6 shows an example of the *CR* tasks identification process for two task graphs. In both cases, we assume that the reconfiguration latency is 4 milliseconds. Note that the tasks are initially mapped onto *Virtual Reconfigurable Units* (VRUs). The reason is that the final mapping of these tasks on the physical Reconfigurable Units (RUs) has not been performed yet (this step will take place in Subsection 3.4). Hence, *VRU1 in Task Graph 1 does not necessarily have to be the same VRU1 in Task Graph 2*.

Continuing with the example, for Task Graph 1, (Figure 6.a), the algorithm firstly checks if, following its initial schedule, its execution time (21 milliseconds) exceeds its deadline (18 milliseconds). This condition is true; hence the algorithm identifies the task which reconfiguration generates the greatest penalty (in this case Task *a*, which penalty is 4 milliseconds), adds it to the *CR_set*, and schedules again Task Graph 1 assuming that Task *a* is already loaded and can be executed without carrying out that reconfiguration. Thus, the new execution time (17 milliseconds) does not exceed the deadline (18 milliseconds). Hence the algorithm stops processing Task Graph 1 and identifies Task *a* as the only *CR* task.

However, for Task Graph 2 (Figure 6.b), the algorithm firstly checks if its execution time (27 milliseconds) does not exceed its deadline (32 milliseconds). This condition is true; hence no task belonging to Task Graph 2 is identified as *CR*.

3.2. Calculation of the "*loading-back factor*"

Once the *CR* tasks have been identified, our methodology assigns each one of them to a different RU. The reason is that we assume that the task graphs can be executed at any point of time, and we know that, according to the *CR* task definition, *all the CR tasks must be already loaded when the task graph starts its execution in order to meet the deadlines*.

Hence we know how many RUs are needed for the *CR* tasks, but we still need to know how many RUs are needed for the remaining tasks (we will name them *non-compulsory-reuse tasks*, or *NCR* tasks). For this purpose, we have developed a simple but powerful technique to determine the number of RUs needed for the *NCR* tasks. The idea is that the *NCR* tasks of a graph can replace some of the *CR* tasks from other graphs, as long as they are loaded them back on time. The number of *CR* tasks that can be replaced is named *loading-back factor*. Thus, for each graph, we define its *loading-back factor* as *the maximum number of new reconfigurations that can be added after the last use of the RUs assigned for non-CR tasks in such a way that the task-graph deadline is still met*. These additional reconfigurations can be used to reload *CR* tasks that were previously replaced.
The algorithm that performs this step is depicted in Figure 7. For each $schedule_i$, it firstly initializes the three variables *possible_rec*, *loading_back_factor* and *marked_tasks* (Lines 2-4). Then, the *while* loop (Lines 5-13) attempts to add new reconfigurations at the end of the execution of the *NCR* tasks in the schedule of the involved task graph. To this end, it firstly selects the task with the greatest idle time after the end of its execution (Line 6), and checks if it is possible to add a new reconfiguration within this time. This can be done if the following two conditions are met:

1. $last\_reconfiguration + rec\_latency \leq deadline_i$: *last_reconfiguration* reports when the last reconfiguration finishes, and *rec_latency* is the reconfiguration latency. Hence this condition guarantees that the reconfiguration circuitry is available for, at least, the *rec_latency* time units before $deadline_i$, i.e. it is possible to add a new reconfiguration after the last one, while meeting the task-graph deadline.
2. $end\ (Task_k) + rec\_latency \leq deadline_i$: it is possible to add a new reconfiguration after the end of the execution of $Task_k$ and still meet the task-graph deadline.

```
1. FOR each schedule_i {
2.   possible_rec := TRUE;
3.   loading_back_factor := 0;
4.   marked_tasks := Ø;
5.   WHILE (last_reconfiguration + rec_latency ≤ deadline_i) AND (possible_rec){
6.     Task_k := select_task (schedule_i, deadline_i, marked_tasks);
7.     IF (end (Task_k) + rec_latency ≤ deadline_i){
8.       update (last_reconfiguration);
9.       mark_task (Task_k , &marked_tasks);
10.      loading_back_factor ++;
11.    }ELSE{
12.      possible_rec := FALSE;
13.  }
14.}
```

Fig. 7. Pseudo-code of the process that identifies the *loading-back factor* of each schedule.

While these two conditions are met, the *while* loop updates the time when the last reconfiguration is carried out (Line 8) and marks the task that was selected in Line 6 (Line 9). This is done in order to avoid selecting the same task twice in this loop, since only one reconfiguration can be added after the end of the execution of each task. Finally the *loading_back_factor* is increased (Line 10).
When the *while* loop finishes, the *loading_back_factor* variable indicates how many reconfigurations can be added, and the *marked_tasks* list reports in which RUs these reconfigurations can be added. This information is used in the third step of this methodology (described in the following subsection).

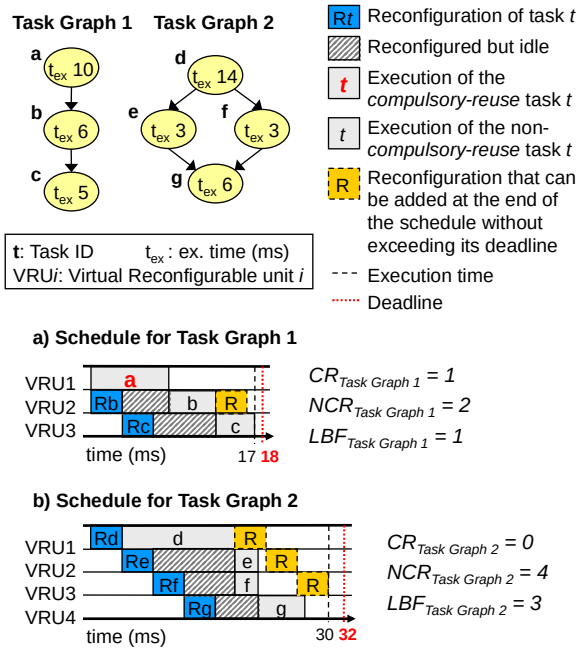

Fig. 8. Example for the *loading-back-factor* calculation. The reconfiguration latency is always 4 milliseconds.

Figure 8 shows an example for the *loading-back factor* calculation for the same two task graphs of Figure 6. Note that in this case, tasks are again mapped onto Virtual Reconfigurable Units (*VRUs*). In this example, the *loading-back factor* of Task Graph 1 is 1. The reason is that it is possible to add a reconfiguration after the end of the execution of Task *b* and still meet the deadline (18 milliseconds). This new reconfiguration is represented in the figure by means of a dashed square labeled with the letter *R*. This is true for Task *b*; however this does not happen with Task *c*, since if a new reconfiguration was added at the end of its execution, the total execution time would be 17+4 = 21 milliseconds, whereas the deadline is 18 milliseconds. Finally, the *loading-back factor* of Task Graph 2 is 3 because 3 new reconfigurations can be added at the end of the execution of Tasks *d*, *e* and *f*, and still meet its deadline (32 milliseconds).

3.3. Calculation of the number of reconfigurable units

Once the *CR* tasks have been identified, we can already provide a first approach to calculate the number of RUs needed to build a reconfigurable system that meets all the task-graph deadlines:

$$Num\_RUs_{first\_approach} = \sum_{i=1}^{n} CR_i + MAX_{i=1}^{n}\{NCR_i\} \quad (1)$$

Where:

1. *n:* the total number of task graphs.
2. CR_i: the number of *CR* tasks of Task Graph *i*.
3. NCR_i: the number of RUs needed for the execution of the non-*CR* tasks from Task Graph *i*.

This solution assigns each *CR* task to a RU, and then it adds the number of RUs needed to execute the remaining ones. For instance, for the task graphs of Figure 8, $Num\_RUs_{first\_approach} = CR_{task\_graph\_1} + CR_{task\_graph\_2} + MAX\ \{NCR_{task\_graph\_1}, NCR_{task\_graph\_2}\} = 1 + 0 + MAX\ \{2, 4\} = \mathbf{5}$.
However, we can further reduce this number by using the *loading-back factor*, which was calculated in the previous step. Thus, once the *loading-back factors* have been computed for all the schedules, the number of RUs that the proposed approach needs to build a system that guarantees that all deadlines are met is:

$$Num\_RUs = \sum_{i=1}^{n} CR_i + MAX_{i=1}^{n}\left\{Extra_{RUs_i}\right\} \quad (2)$$

This formula assigns a RU per *CR* task, and a certain number of additional RUs for the non-*CR* ones. This number is the maximum among all the $Extra_{RUs_i}$ values, which are calculated as follows:

$$Extra_{RUs_i} = NCR_i - MIN\{LBF_i, CR - CR_i\} \quad (3)$$

Where:

1. LBF_i: *loading-back factor* of Task Graph *i*.
2. *CR*: the total number of *CR* tasks in all the task graphs.

According to this formula, the number of extra RUs that are needed for each task graph are the number of *NCR* tasks, minus the term: $MIN\{LBF_i, CR–CR_i\}$. This term indicates the number of *NCR* tasks that can be safely assigned in RUs with *CR* tasks from other task graphs. LBF_i is the *loading-back factor* of Task Graph *i*, whereas $CR–CR_i$ is the number of *CR* tasks that belong to all the task graphs to be scheduled excluding Task Graph *i*. Thus, this term computes the minimum between these two quantities because as many *NCR* tasks from Task Graph *i* as LBF_i can be assigned in RUs also assigned for *CR* tasks for other task graphs (by definition of LBF_i); however, this can be done only if there are enough available RUs with *CR* tasks from other task graphs ($CR–CR_i$). For instance, for the task graphs of Figure 8:

1. $Extra\_RUs_{Task\ Graph\ 1} = 2 - MIN\{1–1, 1\} = 2$.
2. $Extra\_RUs_{Task\ Graph\ 2} = 4 - MIN\{1–0, 3\} = 3$.
3. ***Num_RUs = 1 + 0 + MAX{2, 3} = 4***

In this example, $LBF_{Task\ Graph\ 1}$ is 1, but no RUs with *CR* tasks from another graph are available. Hence this optimization cannot be applied to Task Graph 1. However, $LBF_{Task\ Graph\ 2}$ is 3, and there is 1 RU with a *CR* task from another task graph. Hence, the scheduler can use that RU to reduce the number of additional RUs needed for the *NCR* tasks of Task Graph 2 from 4 to 3. It is interesting to note that the more task graphs are assigned to the system, the more likely is that the scheduler can fully take advantage of the *loading-back factor* in order to reduce the need of additional RUs. Hence as more and more graphs are added, the percentage of RUs used *only* for *NCR* tasks decreases, and more RUs are used for the *CR* tasks.

TABLE I
TASK GRAPHS USED IN THE DEVELOPED EXPERIMENTS

Task graph	Group	Number of tasks	Initial execution time (ms.)	Average task size (slices)	On-demand reconfiguration overhead (ms.)	Number of *CR tasks*	*Loading-back factor*
JPEG	1	4	79	903	16	1	2
MPEG-1	1	5	54	719	20	2	1
Parallel-JPEG	1	8	37	946	32	1	6
HOUGH	1	6	94	611	24	1	2
Pocket GL (1)	2	2	5	831	8	2	0
Pocket GL (2)	2	4	3.3	831	16	0	0
Pocket GL (3)	2	4	14.5	880	16	2	0
Pocket GL (4)	2	5	23	880	20	1	1
Pocket GL (5)	2	5	4.1	1055	20	5	0
Pocket GL (6)	2	5	15.4	999	20	2	0
Pocket GL (7)	2	5	24.8	831	20	2	1
Pocket GL (8)	2	6	24.6	831	24	2	0
Pocket GL (9)	2	6	39.1	831	24	1	4

3.4. Assignment of the tasks to the RUs & Size of the RUs

The last step of this methodology is the assignment of the tasks of all the involved task graphs to the set of available RUs (which number was calculated in the previous step). This assignment takes into account the sizes of the tasks, and tries to minimize the size of the RUs. The assignment algorithm distinguishes between two types of RUs:

1. *CR-RUs*: the set of RUs used to execute *CR* tasks (which can be replaced by *NCR* tasks and loaded back afterwards).
2. *NCR-RUs:* the set of RUs used to execute only non-*CR* tasks.

This assignment is carried out according to the following rules:

1. *CR* tasks:
    a) Each *CR* task is assigned to a different *CR-RU*, and the sizes of these RUs are initialized to the size of the corresponding *CR* task.
2. *NCR* tasks:
    a) The *NCR* tasks are assigned, if possible, to the *NCR-RUs* in order to reduce the number of reconfigurations needed.
    b) If this is not possible, some of them are assigned to the *CR-RUs*, taking advantage of the *loading-back factor*.
    c) In both cases a best-fit policy is applied in order to select one of the available RUs.
    d) However, if the size of the task is bigger than the sizes of all the available RUs, the biggest one is selected, and its size is enlarged.
    e) After a RU is selected, it is not used again for the same task graph.

For this purpose, we have developed a best-fit heuristic algorithm to assign the tasks onto the RUs taking into account these rules. Once the tasks are assigned to the RUs, this algorithm sizes each one of them to the size of the biggest task assigned to it. The algorithm that we have developed is a simple approach in order not to overload too much the system with extra computations. However, any assignment approach could be used in this step, such as an ILP methodology that obtains the optimal solution, as long as its computations do not lead to a significant computational load for the system. In any case, since this is not the main contribution of this paper, we have decided not to describe it in further detail.

In order to be generic, this step uses a simplified area model without taking into account the restrictions of the final target reconfigurable system (some devices are reconfigured only in one dimension (column-based), whether others provide a two-dimensional reconfigurable model (tiled-based), (Steiger, C. et al. [2003])). Our approach can be easily used both for column-based and tiled-based reconfigurable FPGAs. For a column-based FPGA, the area will be the width of the reconfigurable region needed, whereas for a tiled-based FPGA, the area will be the area of the maximum rectangle needed to place all the assigned tasks.

During this process we did not take into account the original shape of the tasks, but only the needed hardware resources. The reason is that in order to use run-time reconfiguration to load a task onto a specific reconfigurable region, that task must be previously synthesized, placed and routed for that specific region. Hence the original task shape will change accordingly to the shape of the region. For instance, in the Xilinx™ design flow the designer must firstly define the reconfigurable regions of the system and then use the PlanAhead tool (XILINX, [2012d]) in order to synthesize all the tasks for their assigned region. In this new synthesis process, this tool is used to obtain an implementation that fits in the given region as long as the region includes the needed hardware resources.

4. EXPERIMENTAL RESULTS

In this section we evaluate the performance of the methodology proposed in this article. For this purpose, we have used a set of task graphs extracted from actual multimedia applications and simulated their execution in a reconfigurable platform. The task graphs that have been used are depicted in Table I: two versions of the JPEG decoder (*JPEG* and *Parallel-JPEG*), a *MPEG-1* encoder, a pattern recognition application (*HOUGH*), and a 3D rendering application based on the open source Pocket-GL library (*Pocket GL (1) – Pocket GL (9)*). In the latter case the application includes 9 different tasks graphs with 2, 4, 5 and 6 consecutive tasks.

For each task graph, the table shows its number of tasks (Column 3) and their initial execution time (Column 4), which represents an ideal scenario with no delays due to the reconfigurations.

Next, Column 5 shows the delays that are generated due to the reconfiguration overheads when using an on-demand approach; i.e. the reconfigurations start when the tasks must be executed. The reconfiguration latency not only depends on the size of the task, but also on the size of the RU. Since we do not know the size of the RUs until the last step of our methodology, and we still have to guarantee that all the deadlines are met; in order to calculate the reconfiguration overhead we apply a worst-case approach. In this case, we assume that the reconfiguration latency is 4 milliseconds, which is the time needed to reconfigure the largest task (1744 slices), at a frequency of 25 MHz in a Virtex-II V6000 FPGA. We have used this device as reference for the calculation of the experimental results in the remainder of this section, both for the reconfiguration overhead and for the task areas.

As the table shows, the reconfiguration overhead generated by an on-demand approach is very significant, especially for the Pocket GL application, where it is even greater than its initial task-graph execution time in 5 out of the 9 evaluated task graphs. Finally, Column 6 shows the number of *CR* tasks of each task graph and Column 7 does likewise with their *loading-back factors* in the most demanding scenario: in this case the deadline is the initial execution time (i.e., the reconfigurations cannot introduce any delay, otherwise the deadlines of the task graphs are not met).

Since the actual impact of the reconfigurations heavily depends on the task graphs, we have divided our task graphs of Table I in two groups. The first group includes the JPEG, MPEG, Hough and Parallel-JPEG graphs, whereas the second group includes all the Pocket-GL ones. In both groups, the reconfigurations are critical for the performance, but

in the second one their impact is clearly greater since the average execution time of their tasks is smaller (only 3.66 milliseconds vs. 11.48 milliseconds on average for tasks from Group 1). As a result, only 20% of the tasks from Group 1 belong to the *CR* set, whereas almost 50% of the tasks of the second group belong to that category. Moreover, the average *loading-back factor* for the tasks belonging to Group 1 is 3, whereas for Group 2, it is just 1.

4.1. Number of reconfigurable units used

In a first experiment we have evaluated the resources consumption of the system when the presented approach is used for the execution of the applications shown in Table I, and meeting different *extended deadlines*. This parameter represents the maximum acceptable delay with respect to the optimal execution time of the applications, in such a way that: *execution time = extended deadline + optimal execution time*.

In all the experiments presented in this section we have compared the resources consumption of our approach (labeled as "*CR+LB approach*" in all the figures below) with the first one that we previously introduced in Formula (1) ("*CR approach*", described in Subsection 3.3). With this comparison we can identify the benefits of taking advantage of the *loading-back factor*. In addition, we have also compared our approach with a static one ("*Static approach*") that does not apply run-time reconfiguration to reduce the area requirements. Hence in the latter all the tasks of all the involved task graphs are executed in separate RUs.

First of all, Figure 9 shows the number of RUs that are needed when the *extended deadlines* of all the involved applications range from 0 to 4 milliseconds. The plots of Figures 9.a and 9.b refer to the resources consumption of the system when executing *all the task graphs* belonging to Group 1 and Group 2, respectively.

These results show that the *CR approach* already greatly reduces the resources consumption obtained by the static one. Thus, it achieves 61.49% and 71.77% of average resources savings for the task graphs from Groups 1 and 2, respectively. For instance, Figure 9.b shows that our approach needs just 6 RUs to execute 9 task graphs from Group 2 when the *extended deadlines* of the applications is 4 milliseconds. This greatly differs from the 42 RUs that are needed when the *Static approach* is used. Thus, we can see that these 6 RUs are shared among all the tasks that are executed (42 tasks).

However, the proposed *CR+LB approach* further improves these nice results, since it achieves 65.22% and 75.17% of resources savings on average with respect to the static one for the tasks from Groups 1 and 2, respectively.

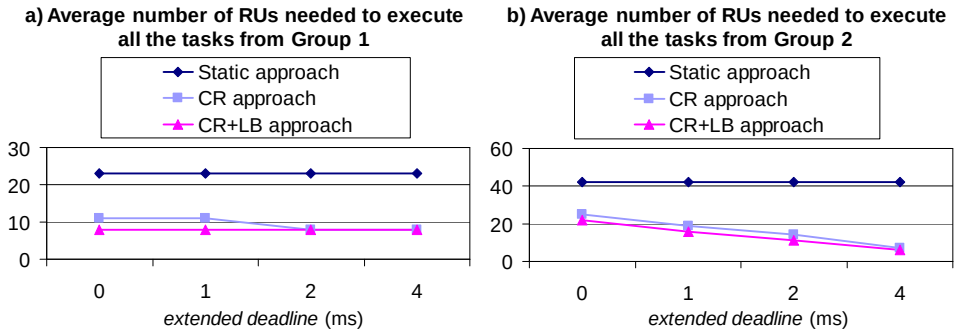

Fig. 9. Average number of required reconfigurable units when applying the proposed approach with different values of the *extended deadline*, in comparison with a static approach.

In this figure we can also observe that, for the tasks from Group 2, the number of RUs needed by both the *CR* and *CR+LB* methodologies decreases as the *extended deadline* increases. The reason is that as this parameter grows, the *loading-back factor* also grows, and the number of *CR tasks* in the task graphs decreases.
Thus, for instance, Figure 9.b shows that the number of RUs needed by the *CR+LB approach* decreases from 25 to just 6 when the *extended deadline* ranges from 0 to 4 milliseconds, respectively.
In a second experiment we have evaluated the average amount of RUs that are needed for a variable number of different task graphs that are executed altogether. Figure 10 shows these results. As in Figure 9, Figures 10.a and 10.b refer to the results obtained when executing the applications from Groups 1 and 2, respectively. For Group 1 the number of task graphs that are executed altogether ranges from 2 to 4 since this group contains only 4 task graphs. However, for Group 2 (Figure 10.b) this number ranges from 2 to 9. In all the cases, the figure shows the average results of the experiments corresponding to all the possible combinations that contain the same number of task graphs.

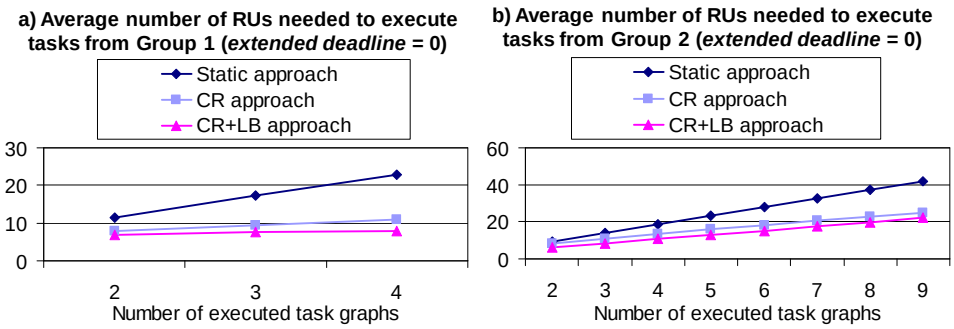

Fig. 10. Average number of required reconfigurable units when applying the proposed approach for different number of executed task graphs, in comparison with a static approach.

As it is expected, the results of this experiment show that if the number of different tasks graphs increases, more RUs are needed. However, it is important to point out that the reduction on the number of RUs obtained by our *CR* and *CR+LB* approaches also increases as more different graphs are executed. Hence, this shows that these techniques improve the scalability of the system.

4.2. Number of FPGA slices consumed

In this subsection, we have repeated the two experiments described in the previous one, but this time evaluating the amount of FPGA slices that are needed. Figure 11 shows the amount of slices used for the same experiments carried out in Figure 9, whereas Figure 12 does likewise for the experiments of Figure 10.
In order to carry out these experiments, we have used implementation results when the evaluated task graphs were synthesized in a Virtex-II V6000 FPGA.

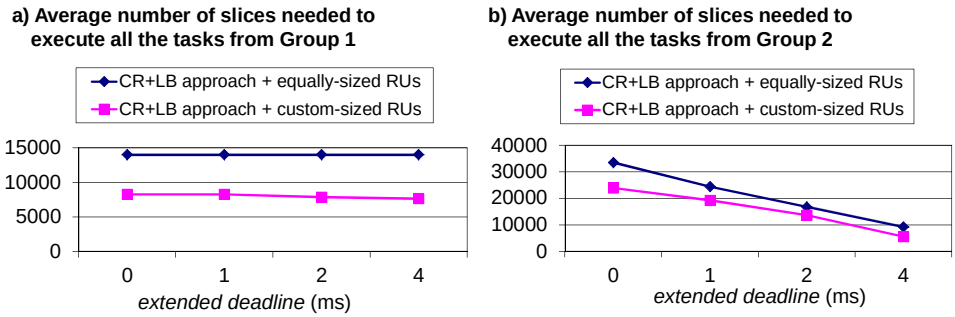

Fig. 11. Average amount of required Virtex-II FPGA slices when applying the proposed approach with different values of the *extended deadline*.

In both cases we have compared the *CR+LB* methodology with and without applying the best-fit technique in the step "*Assignment of the tasks in the RUs & size of the RUs*" (which was previously described in Subsection 3.4). These results have been labeled in the figures as "*CR+LB approach + custom-sized RUs*" and "*CR+LB approach + equally-sized RUs*", respectively. In the former case, the RUs are dimensioned following the algorithm mentioned in Subsection 3.4. And in the latter case, the size of all the RUs needed is simply the size of the largest task executed in the system. This second approach is commonly found in most reconfigurable systems that make the reconfiguration decisions at run time. Some examples are research works presented by Clemente, J.A. et al. [2010], Noguera, J. et al. [2002], [2004] and Ghiasi, S. et al. [2004].

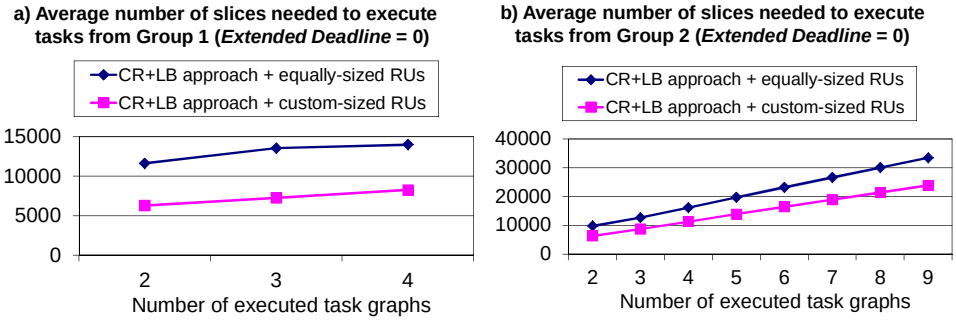

Fig. 12. Average amount of required Virtex-II FPGA slices when applying the proposed approach for different numbers of executed task graphs.

As these figures show, our mapping technique leads to very important area savings. The reason is that we assign the tasks to the RUs trying to minimize the area needed. Since this process is carried out at design time, it is still possible to adjust their area and to achieve these important savings.

The results depicted in these figures do not include the additional slices needed to communicate a RU with the system. However, this overhead is typically negligible. For instance, in a Virtex-5 FPGA only 72 additional slices are needed in order to connect a RU to the system bus.

It is also important to mention that, although our methodology customizes the size of each RU according to the maximum of the sizes of the tasks assigned to them, some internal fragmentation may still appear, since the tasks assigned to the same RU can have different sizes. According to our measurements, an average 9.3% and 2.1% of the RU

resources is wasted for the experiments regarding tasks from Groups 1 and 2, respectively. We think that these results are affordable. In any case, if for other task sets the fragmentation is a problem, a more complex assignment algorithm could be used instead to carry out this step. The assignment algorithm can be easily replaced since it is the last step of our methodology and it is orthogonal to the previous ones.

4.3. Effects in the variation of the reconfiguration latency

In this subsection we evaluate the proposed methodology for different reconfiguration latencies. When this latency grows, the ratio between this parameter and the execution time of the task graphs also grows. Hence this leads to variations in their number of *CR* tasks and their *loading-back factor*.

Figure 13 shows the number of RUs needed when the *Static*, *CR* and *CR+LB* approaches are applied for the execution of the task graphs from Groups 1 and 2, respectively, and for different values of reconfiguration latency (from 2 to 8 milliseconds). As in the previous experiment, the *extended deadlines* of all the task graphs are 0.

Both in Figures 13.a and 13.b we can observe that the *CR* and *CR+LB* approaches again greatly outperform the static one, and that *CR+LB* works always better than *CR*. However, the impact of the evaluated variations in the reconfiguration latency differs for the execution of the task graphs from Groups 1 and 2, respectively. In fact, when the reconfiguration latency is multiplied by 4, the number of needed RUs grows from 8 to 13 (+62.5%) for task graphs from Group 1. However, for Group 2, this variation ranges from 22 to 41 (+86.36%) and, in the worst case, it is almost impossible to apply partial reconfiguration without incurring in any overhead (note that in this case 41 RUs are needed, whereas the total number of different tasks is 42).

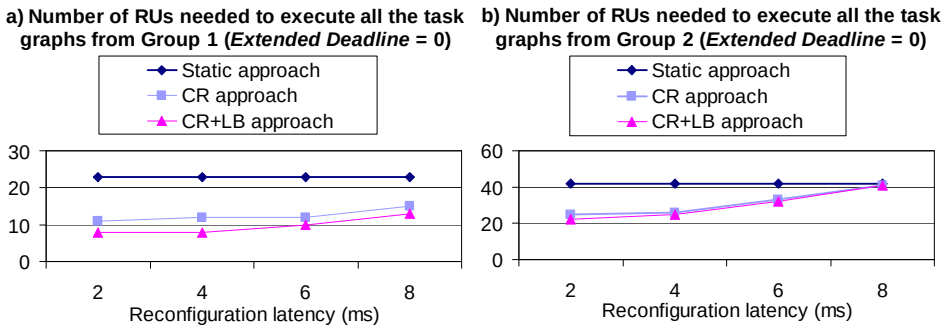

Fig. 13. Number of required reconfigurable units when applying the proposed approach for tasks with different reconfiguration latencies, in comparison with a static approach.

The reason of this change is that, in the latter case, the reconfiguration latency is much greater than the execution time of the tasks. Indeed, for the task graphs from Group 1, the ratio between these two parameters ranges from 0.15 to 0.6 when the reconfiguration latency ranges from 2 to 8 milliseconds; whereas for the tasks from Group 2, this ratio ranges from 0.8 to 3.6. Hence, in the latter case the only way to completely eliminate the reconfiguration overhead is by adding more and more RUs.

Therefore, in this figure we can see that our methodology adapts well to very demanding scenarios in which the execution times of the applications are close to their average reconfiguration latency. The area savings are not significant *only* when the reconfiguration latency is clearly greater than the average execution time of the tasks. If

this is the case, the only way of guarantee that the reconfigurations do not introduce any delay in the system is to carry out very few of them. For that reason, the results of our approach and the static one converge for the graphs in Group 2 when the reconfiguration latency is 8 milliseconds.

4.4. Comparison with other best-effort approaches

Finally, in this subsection we compare the performance of the *CR+LB approach* with two representative best-effort schedulers. These schedulers apply two techniques to reduce the reconfiguration overhead. On the one hand, they attempt to hide the reconfiguration latency by applying a task-graph prefetch technique. *In this case we have used a scheduler based on a branch and bound approach that guarantees the optimal best-effort schedule of the reconfigurations*. On the other hand, they apply a replacement technique that attempts to maximize task reuse. Each scheduler applies a different replacement technique:

1. LRU (*Least Recently Used*), which replaces the reconfigurable unit that contains the task that was the least recently used with respect to the remaining ones.
2. LFD (*Longest Forward Distance*), which replaces the reconfigurable unit that contains the task that will be requested farthest into the future. This implies that, in order to use it, the system needs to know the complete sequence of task graphs that will be executed in the system. Hence, it cannot be applied in the dynamic environment of this paper, in which we assume that the task graphs come for their execution in an unpredictable way. However, it can be used as a reference for a particular experiment where the sequence of task graphs to be executed is known in advance. This policy was originally proposed by Belady, L.A. [1966] and guarantees the optimal reuse rate.

Figure 14 shows the percentages of hard deadlines (when the *extended deadline* of all the task graphs is 0) that are missed when the task graphs from Groups 1 and 2 are executed using this scheduler. In all these experiments we have executed a sequence of 1000 task graphs randomly selected among the set of involved task graphs in Table I.
Both Figures 14.a and 14.b show what happens when the reconfiguration latency ranges from 2 to 8 milliseconds. In each one of these cases, the number of RUs that the system includes is the amount of RUs that our approach needs to guarantee that the temporal constraints of the given set of task graphs are always met (see the results of Figure 13). *Hence in all these cases our approach meets all the deadlines.* Thus, for the experiments regarding Group 1 (Figure 14.a), the system includes 8, 8, 10 and 13 RUs when the reconfiguration latency is 2, 4, 6 and 8 milliseconds, and for Group 2 (Figure 14.b), 23, 25, 32 and 41 RUs are included for each case, respectively.

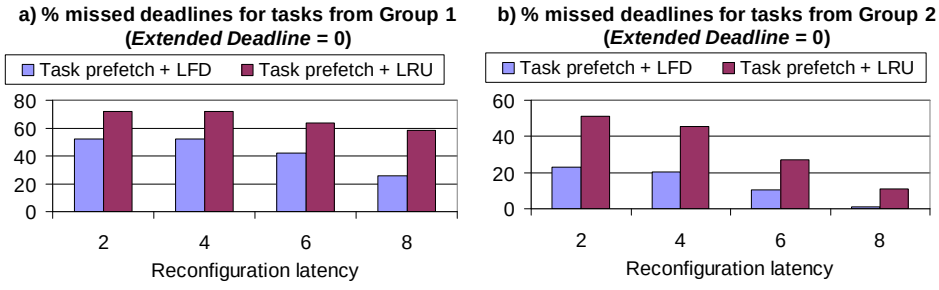

Fig. 14. Percentage of missed hard deadlines when executing task graphs from Groups 1 (a) and 2 (b), and using a scheduler that applies prefetch and different replacement techniques (LRU and LFD)

In the figure, we can observe that the *Task Prefetch + LRU* approach misses many deadlines. Thus, for the task graphs from Group 1 and Group 2, it misses on average 66.5% and 33.7% of the task-graph deadlines, which are very high rates. The *Task Prefetch + LFD* approach works better, since these deadline misses decrease to 43% and 13.8%, respectively.
However, in both cases these results are unacceptable, especially in hard real-time systems, when the only acceptable result is that *all* the task-graph deadlines are met. On the contrary, with the same number of RUs, our methodology ensures that 100% of these deadlines are met. This does not mean that our work is better than a best-effort approach; it is different since it targets a different objective. Best-effort approaches are designed to maximize the average throughput, but some deadlines can be missed. On the contrary, real-time systems require a different approach, as it has been shown in Figure 14.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have proposed a methodology to develop reconfigurable systems that can execute a set of given applications, guaranteeing that all their hard real-time constraints are always met. For this purpose, this methodology analyzes the applications (represented as DAGs) and their deadlines; and determines the size of the reconfigurable system, trying to minimize the resources consumption. In addition, this methodology maps the tasks on the reconfigurable resources and generates the final schedules, thereby allowing to transparently manage the reconfigurations of the tasks from the user's point of view. The results have shown that the proposed approach achieves important resources savings with respect to an equivalent static solution. In addition, this work opens the possibility of using run-time reconfigurations in a hard real-time context, since it guarantees that 100% of the task-graph deadlines are met, whereas other scheduling best-effort or soft real-time techniques proposed in the literature cannot ensure this point with the same amount of reconfigurable resources.
As future work, we would like to extend this approach to reduce the internal fragmentation of the tasks in the RUs.

REFERENCES

ALTERA, 2011. http://www.altera.com/products/devices/stratix-fpgas/stratix-v/stxv-index.jsp.

ALTERA, 2011. https://www.altera.com/download/software/quartus-ii-we.

ALTERA, 2011. http://www.altera.com/products/ip/processors/nios2/ni2-index.html.

BANERJEE, S., BOZORGZADEH, E., AND DUTT, N., 2009. Exploiting Application Data-Parallelism on Dynamically Reconfigurable Architectures: Placement and Architectural Considerations. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 17, 2, 234-247.

BELADY, L.A., 1966, A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5, 78-101.

CHANG, C., WAWRZYNEK, J., AND BRODERSEN, R.W., 2005. BEE2: A High-End Reconfigurable Computing System. *IEEE Design & Test of Computers*, 22, 2, 114-125.

CLEMENTE, J.A., GONZÁLEZ, C., RESANO, J., AND MOZOS, D., 2010. A Task Graph Execution Manager for Reconfigurable Multi-tasking Systems. *Microprocessors and Microsystems: Embedded Hardware Design (MICPRO)*, 34, 2-4, 73-83.

IBM MICROELECTRONICS, 1999. *CoreConnect Bus Architecture, A 32-, 64-, 128-bit core on-chip bus standard*.

CORDONE, R., REDAELLI, F., REDAELLI, M.A., SANTAMBROGIO, M.D., AND SCIUTO, D., 2009. Partitioning and Scheduling of Task Graphs on Partially Reconfigurable FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 28, 5, 662-675.

DANNE, K.D., AND PLATZNER, M., 2005. A Heuristic Approach to Schedule Periodic Real-Time Tasks on Reconfigurable Hardware. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, Tampere, Finland, 258-573.

DANNE, K.D., MIIHLENBERND, R., AND PLATZNER, M., 2006. Executing Hardware Tasks on Dynamically Reconfigurable Devices under Real-Time Conditions. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, Madrid, Spain, 1-6.

DITTMAN, F., AND FRANK, S., 2007. Hard Real-Time Reconfiguration Port Scheduling. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE)*, Nice, France, 34-40.

EDK, 2010. http://www.xilinx.com/support/documentation/dt_edk_edk12-3.htm

FAZLALI, M., SABEGHI, M., ZAKEROLHOSSEINI, A., AND BERTELS, K., 2010. Efficient Task Scheduling for Runtime Reconfigurable Systems. *Journal of Systems Architecture (JSA)*, 56, 11, 623-632.

GHIASI, S., NAHAPETIAN, A., AND SARRAFZADEH, M., 2004. An Optimal Algorithm for Minimizing Run-Time Reconfiguration Delay. *ACM Transactions on Embedded Computing Systems (TECS)*, 3, 2, 237-256.

GÖHRINGER, D., HÜBNER, M., ZEUTEBOUO, E.N., AND BECKER, J., 2011. Operating System for Runtime Reconfigurable Multiprocessor Systems. *International Journal of Reconfigurable Computing (IJRC)*, 2011, 1-16.

HAUCK, S., 1998. Configuration Prefetch for Single Context Reconfigurable Coprocessors. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, Monterey, CA, USA, 65-78.

KAVI, K., BUCKLES, B., AND BHAT, U., 1986. A Formal Definition of Data Flow Graph Models. *IEEE Transactions on Computers*, C-35, 11, 940-948.

KOOTI, H., BOZORGZADEH, E., LIAO, S., AND BAO, L, 2010. Transition-aware Real-Time Task Scheduling for Reconfigurable Embedded Systems. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE)*. Dresden, Germany, 232-237.

LI, Z., AND HAUCK, S., 2002. Configuration Prefetching Techniques for Partial Reconfigurable Coprocessors with Relocation and Defragmentation. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, Monterey, CA, USA, 187-195.

LÜBBERS, E., AND PLATZNER, M., 2007. ReconOS: an RTOS supporting hard- and software threads. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, Amsterdam, the Netherlands, 441-446.

NAHAPETIAN, A., BRISK, P., GHIASI, S., AND SARRAFZADEH, M., 2009. An approximation algorithm for scheduling on heterogeneous reconfigurable resources. *ACM Transactions on Embedded Computing Systems (TECS)*, 9, 1, 1-20.

NOGUERA, J., AND BADÍA, R.M., 2002. Dynamic Run-Time Hardware/Software Scheduling Techniques for Reconfigurable Architectures. In *Proceedings of the International Symposium on Hardware/Software Codesign (CODES)*, Estes Park, CO, USA, 205-210.

NOGUERA, J., AND BADÍA, R.M., 2004. Multitasking on Reconfigurable Architectures: Microarchitecture Support and Dynamic Scheduling. *ACM Transactions on Embedded Computing Systems (TECS)*, 3, 2, 385-406.

OPENCORES, 2011. http://opencores.org/newsletter,2011,02,#n5.

PAN, Z., AND WELLS, B.E., 2008. Hardware Supported Task Scheduling on Dynamically Reconfigurable SoC Architectures. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 16, 11, 1465-1474.

SIM, J.E., WONG, W.F., AND TEICH, J., 2009. Optimal Placement-aware Trace-Based Scheduling of Hardware Reconfigurations for FPGA Accelerators. In *Proceedings of the 17th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, USA, 279-282.

SO, H. K.-H., AND BRODERSEN, R., 2008. File System Access from Reconfigurable FPGA Hardware Processes in BORPH. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, Heidelberg, Germany, 567-570.

SO, H. K.-H., AND BRODERSEN, R., 2008. Runtime File System Support for Reconfigurable FPGA Hardware Processes in BORPH. In *Proceedings of the 16th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Palo Alto, CA, USA, 285-286.

STEIGER, C., WALDER, H., PLATZNER, M., AND THIELE, L, 2003. Online Scheduling and Placement of Real-Time Tasks to Partially Reconfigurable Devices. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS)*, Cancun, Mexico, 224-225.

SPRUNT, B., SHA, L, AND LEHOCZKY, J., 1989. Aperiodic Task Scheduling for Hard-Real-Time Systems. *Real-Time Systems*, 1, 1, 27-60.

XILINX, 2005. *Local Memory Bus (LMB) v1.0 (v1.00a).*

XILINX, 2009. *PlanAhead User Guide*, UG632 (v11.4).

XILINX, 2012a. *MicroBlaze Processor Reference Guide*, UG081 (v13.4).

XILINX, 2012b. *Virtex-5 FPGA User Guide*, UG190 (v5.4).

XILINX, 2012c. *Virtex-6 Family Overview*, DS150 (v2.4).

XILINX, 2012d. *Partial Reconfiguration User Guide*, UG702 (v 14.1).